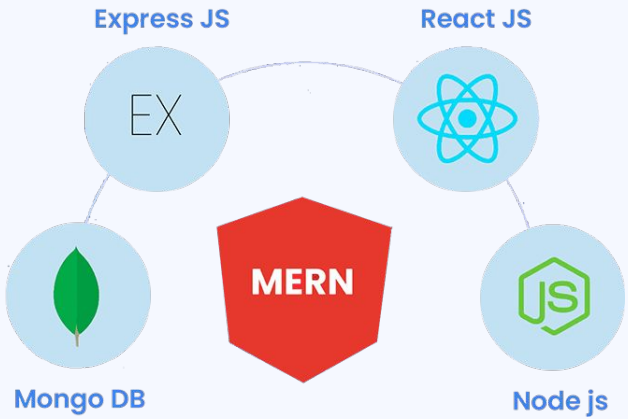


Full Stack Development Using MERN

Day 04 - 12th May 2024

Lecturer Details:

Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

Prameeth Madhuwantha
bap@ucsc.cmb.ac.lk
0772888098

Lecture 04

Introduction to Node.js

Overview of Node.js

What is Node.js?

Node JS is an open-source and cross-platform runtime environment built on Chrome's V8 JavaScript engine for executing JavaScript code outside of a browser

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

Why to learn Node JS?

Common task for a web server can be to open a file on the server and return the content to the client.

PHP handles a file request:

- Sends the task to the computer's file system.
- Waits while the file system opens and reads the file.
- Returns the content to the client.
- Ready to handle the next request.

Why to learn Node JS?

Node.js handles a file request:

- Sends the task to the computer's file system.
- Ready to handle the next request.
- When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

Components of Node.js

- **Console**: Provides a simple way to write to the stdout and stderr streams, useful for logging and printing outputs.
- **Debugger**: Node.js has a built-in debugger that can be used to debug JavaScript code executed by Node.
- **Error Handling**: Mechanism to handle errors gracefully in Node.js applications, ensuring stability and debugging support.
- **Domain**: Deprecated module in Node.js used to intercept unhandled errors and manage multiple I/O operations.
- **Call Backs**: A core concept in Node.js where functions are passed as arguments to other functions and are executed asynchronously.

Components and Functionalities

- **Buffer**: Provides a way to work with raw binary data directly in memory, without having to convert to strings.
- **Modules**: Code encapsulation mechanism in Node.js. Modules allow you to break down your code into reusable parts.
- **DNS**: Node.js provides DNS module to perform domain name resolution operations.
- **Cluster**: Allows you to create child processes (workers) that share the same server port, facilitating scalability on multi-core systems.
- **Net**: Provides an asynchronous network API to create stream-based TCP or IPC servers and clients.

How does Node.js work?

- Node.js accepts the request from the clients and sends the response, while working with the request node.js handles them with a single thread.
- To operate I/O operations or requests node.js use the concept of threads.
- Thread is a sequence of instructions that the server needs to perform. It runs parallel on the server to provide the information to multiple clients.
- Node.js is an event loop single-threaded language. It can handle concurrent requests with a single thread without blocking it for one request.

Advantages of Node.js

- Real-time web apps: Node.js is much more preferable because of faster synchronization. Also, the event loop avoids HTTP overloaded for Node.js development.
- Fast Suite: NodeJS acts like a fast suite and all the operations can be done quickly like reading or writing in the database, network connection, or file system
- Advantage of Caching: It provides the caching of a single module. Whenever there is any request for the first module, it gets cached in the application memory, so you don’t need to re-execute the code.
- Data Streaming: In NodeJS HTTP request and response are considered as two separate events.

Starting with Node.js

With Node.js we can either create a console based application or a web based application.

1. Console Based Application
    - Console based applications are run using Node.js command prompt
    - Console module in Node.js provide a simple debugging console.
1. Web-based Node.js Application
    - HTTP Module: Built-in HTTP module that allows the creation of HTTP server without the use of external libraries
    - Express.js Integration: Express.js is a popular web application framework for Node.js.

Console Based Application Example 1

- Create a file - hello.js

```
JS hello.js
  1   console.log("Hello wrold");
```

```
PS D:\CSE Course> node hello.js
Hello wrold
PS D:\CSE Course>
```

Example 2 (JS code to take user input)

```
console.log(process.argv.slice(2));
```

```
PS D:\CSE Course> node hello.js Hello World
[ 'Hello', 'World' ]
PS D:\CSE Course>
```

Basics with Node.js

Node.js contains various types of data types similar to JavaScript:

- **Boolean**: Represents a true or false value.
- **Undefined**: A variable that has been declared but hasn't been assigned a value is undefined.
- **Null**: Represents a deliberate absence of any value.
- **String**: Represents textual data.
- **Number**: Represents numeric values.
- **Loose Typing**: Node.js supports loose typing, which means you don’t need to specify what type of information will be stored in a variable in advance. In Node.js, you can use various keywords to declare variables:
    - var, let, const

Basics with Node.js Contd

- **var**: It's the oldest way to declare variables. However, its scope can sometimes be confusing and is not recommended for use in modern applications.
- **let**: Introduced in ES6, it allows block-scoped variable declaration, which means the variable's scope is limited to the block, statement, or expression where it's defined.
- **const**: Also introduced in ES6, it's similar to let but is used to declare variables whose values should never be reassigned. It effectively makes the variable a constant, offering an additional layer of protection against unintended changes.

Node.js Basics Example 1

```
// Variable store number data type
let a = 35;
console.log(typeof a);

// Variable store string data type
a = "GeeksforGeeks";
console.log(typeof a);

// Variable store Boolean data type
a = true;
console.log(typeof a);

// Variable store undefined (no value) data type
a = undefined;
console.log(typeof a);
```

```
PS C:\Users\Anushka\Downloads\react\node.js> node first.js
number
string
boolean
undefined
```

Node.js Basics Example 2

```
// Using var
var fname = "John";
console.log(fname); // Outputs: John

// Using let
let age = 25;
console.log(age); // Outputs: 25

// Using const
const PI = 3.141592653589793;
console.log(PI); // Outputs: 3.141592653589793
```

```
John
25
3.141592653589793
```

Remember that while you cannot reassign a new value to a constant, if it holds an object, you can still modify the properties of that object.

Objects & Functions in Node.js

Objects: Node.js objects are the same as JavaScript objects i.e. the objects are similar to variables and it contains many values which are written as name: value pairs. Name and value are separated by a colon and every pair is separated by a comma.

Functions: Node.js functions are defined using the function keyword then the name of the function and parameters which are passed in the function. In Node.js, we don't have to specify datatypes for the parameters and check the number of arguments received. Node.js functions follow every rule which is there while writing JavaScript functions.

Objects Example

```
let company = {
    Name: "GeeksforGeeks",
    Address: "Noida",
    Contact: "+919876543210",
    Email: "abc@geeksforgeeks.org"
};

// Display the object information
console.log("Information of variable company:", company);

// Display the type of variable
console.log("Type of variable company:", typeof company);
```

```
Information of variable company: {
  Name: 'GeeksforGeeks',
  Address: 'Noida',
  Contact: '+919876543210',
  Email: 'abc@geeksforgeeks.org'
}
Type of variable company: object
```

Functions Example

```
function multiply(num1, num2) {

    // It returns the multiplication
    // of num1 and num2
    return num1 * num2;
}

// Declare variable
let x = 2;
let y = 3;

// Display the answer returned by
// multiply function
console.log("Multiplication of", x,
    "and", y, "is", multiply(x, y));
```

```
Multiplication of 2 and 3 is 6
```

- Create a Node.js function that returns the sum of two numbers using an object to organize related functions?

- Implement a simple bank account system in Node.js
  - Define a function that creates and returns a bank account object
  - Function accepts parameters of owner name and initial balance
  - Each bank account object will have properties for owner name and balance, as well as methods for depositing, withdrawing, and checking the balance.

```
// Define an object to hold math-related functions
const mathOperations = {
  // Function to add two numbers
  add: function (num1, num2) {
    return num1 + num2;
  },

  // Function to subtract two numbers
  subtract: function (num1, num2) {
    return num1 - num2;
  },

  // Function to multiply two numbers
  multiply: function (num1, num2) {
    return num1 * num2;
  },

  // Function to divide two numbers
  divide: function (num1, num2) {
    return num1 / num2;
  }
};

// Example usage
const sum = mathOperations.add(5, 3); // returns 8
console.log("Sum:", sum);
```

```
// Function to create a new bank account object
function createBankAccount(ownerName, initialBalance) {
  let account = {
    ownerName: ownerName,
    balance: initialBalance,

    // Method to deposit money into the account
    deposit: function(amount) {
      account.balance += amount; // Directly access `account.balance` instead of using `this`
      console.log(`Deposit of $${amount} successful. New balance: $${account.balance}`);
    },

    // Method to withdraw money from the account
    withdraw: function(amount) {
      if (amount <= account.balance) { // Directly access `account.balance` instead of using
`this`
        account.balance -= amount;
        console.log(`Withdrawal of $${amount} successful. New balance:
$${account.balance}`);
      } else {
        console.log("Insufficient funds!");
      }
    },

    // Method to check the balance
    checkBalance: function() {
      console.log(`Current balance for ${account.ownerName}: $${account.balance}`); //
Directly access `account.ownerName` and `account.balance` instead of using `this`
    }
  };

  return account;
}

// Example usage
const account1 = createBankAccount("Jane Doe", 1000);
account1.deposit(500); // Deposit $500
account1.withdraw(200); // Withdraw $200
account1.checkBalance(); // Check balance
```

Scope in Node.js

Scope in Node.js (as in JavaScript) dictates where variables, functions, and objects are accessible. Scope can either be local or global:

- Local Scope (or Function Scope): Variables declared inside a function using the var, let, or const keywords are only accessible within that function. They can't be accessed outside that function.
- Global Scope: Variables declared outside a function become globally accessible throughout your application. However, it's generally not a good practice to clutter the global scope.

Local Scope Example

```
function showLocalScope() {
    let localVariable = "I'm a local variable";
    console.log(localVariable); // Outputs: I'm a local variable
}

showLocalScope();
// console.log(localVariable);
// Uncommenting this will throw an error because
// localVariable is not accessible outside the function.
```

```
PS C:\Users\Anushka\Downloads\react\node.js> node scope.js
I'm a local variable
```

Global Scope Example

```
let globalVariable = "I'm a global variable";

function showGlobalScope() {
    console.log(globalVariable); // Outputs: I'm a global variable
}

function modifyGlobalScope() {
    globalVariable = "I've been modified!";
    console.log(globalVariable); // Outputs: I've been modified!
}

showGlobalScope();
modifyGlobalScope();
console.log(globalVariable); // Outputs: I've been modified!
```

```
I'm a global variable
I've been modified!
I've been modified!
```

Closures in Node.js

Closures are a fundamental concept in JavaScript, including its server-side runtime environment, Node.js. They allow a function to access variables from an enclosing scope or environment even after the outer function has executed and returned. This behavior is intrinsic to how lexical scoping works in JavaScript.

How Closures Work:

When you define a function inside another function and expose this inner function, either by returning it or passing it to another function, the inner function will have access to the variables in the outer function even after the outer function has finished executing

Closures Example

```
function outerFunction() {
    let outerVariable = 'I am an outer variable';

    function innerFunction() {
        console.log(outerVariable); // innerFunction can access outerVariable
    }

    return innerFunction;
}

let myClosure = outerFunction(); // Returns the innerFunction
myClosure(); // Outputs: 'I am an outer variable'
```

```
I am an outer variable
```

Practical Uses of Closures

Data Privacy: Closures can be used to emulate private methods or properties in JavaScript.

Event Handlers: They are often used in JavaScript for event handlers and callbacks to capture context or state.

Functional Programming: Higher-order functions like map, filter, and reduce make use of closures to maintain state.

Timeouts and Intervals: When using setTimeout or setInterval, closures help in accessing the outer function's context.

Example: Data Privacy with Closures

```
function createCounter() {
    let count = 0; // This is a private variable

    // We return an increment function
    return function increment() {
        count++;
        return count;
    };
}

// Get the increment function with access to the 'count' variable
let counter = createCounter();

console.log(counter()); // Outputs: 1
console.log(counter()); // Outputs: 2
```

```
PS C:\Users\Anushka\Downloads\react\node.js> node data_privacy.js
1
2
```

JavaScript Arrow Function

Before Arrow

```
hello = function() {
  return "Hello World!";
}
```

With Arrow

```
hello = () => {
  return "Hello World!";
}
```

Arrow Functions Return Value by Default

```
hello = () => "Hello World!";
```

Arrow Functions With Parameters

```
hello = (val) => "Hello " + val;
```

If you have only one parameter, you can skip the parentheses as well:

```
hello = val => "Hello " + val;
```

Arrays and Array Methods

Arrays in Node.js are ordered collections and can hold multiple values of different data types. There are several built-in methods in Node.js to manipulate arrays:

- **push()**: Adds one or more elements to the end of an array and returns the new length.
- **pop()**: Removes the last element from an array and returns that element.
- **shift()**: Removes the first element from an array and returns that element.

Arrays and Array Methods Contd

- **unshift()**: Adds one or more elements to the beginning of an array and returns the new length.
- **map()**: Creates a new array by calling a function on every array element.
- **filter()**: Creates a new array with every element that passes the test in a given function.
- **reduce()**: Reduces the array to a single value by processing each value from left to right.

Arrays and Array Methods Example

1. push()

```
let fruits = ['apple', 'banana'];
let newLength = fruits.push('orange', 'grape');
console.log(fruits); // Outputs: ['apple', 'banana', 'orange', 'grape']
console.log(newLength); // Outputs: 4
```

```
[ 'apple', 'banana', 'orange', 'grape' ]
4
```

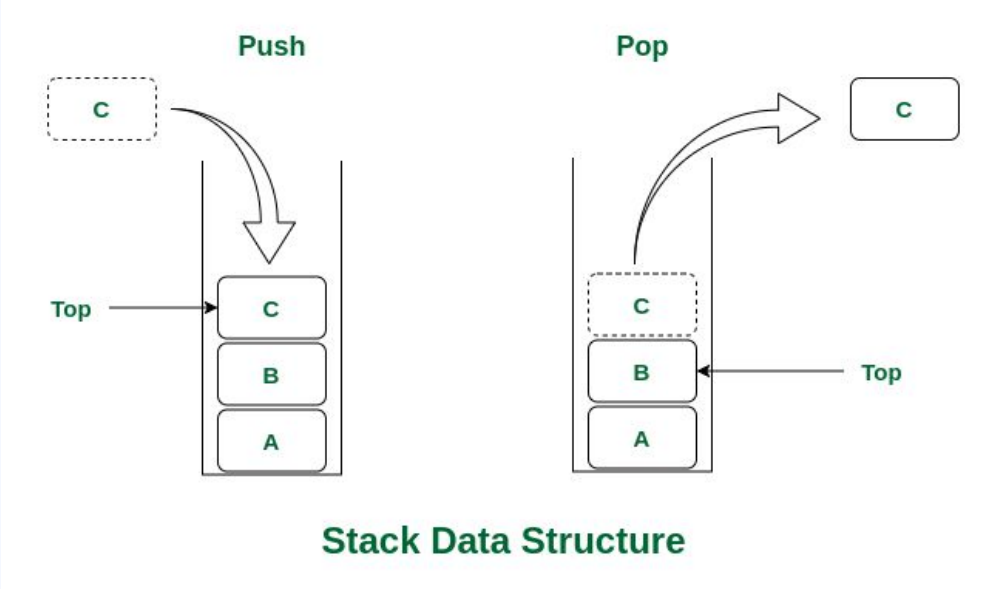
1. pop()

```
let fruits = ['apple', 'banana', 'orange'];
let lastFruit = fruits.pop();
console.log(fruits); // Outputs: ['apple', 'banana']
console.log(lastFruit); // Outputs: 'orange'
```

```
[ 'apple', 'banana' ]
orange
```

A practical example :

Implement a stack data structure in Node.js using the push() and pop() methods.



Stack Data Structure

```
// Initialize an empty stack
let stack = [];

// Function to push an element onto the stack
function pushToStack(element) {
  stack.push(element);
  console.log(`Pushed ${element} onto the stack.`);
}

// Function to pop an element from the stack
function popFromStack() {
  if (stack.length === 0) {
    console.log("Stack underflow: Cannot pop from an empty
stack.");
    return null;
  } else {
    const poppedElement = stack.pop();
    console.log(`Popped ${poppedElement} from the stack.`);
    return poppedElement;
  }
}

// Example usage
pushToStack(10); // Push 10 onto the stack
pushToStack(20); // Push 20 onto the stack
pushToStack(30); // Push 30 onto the stack
popFromStack(); // Pop an element from the stack
popFromStack(); // Pop another element from the stack
```

Arrays and Array Methods Example

3. shift()

```
let fruits = ['apple', 'banana', 'orange'];
let firstFruit = fruits.shift();
console.log(fruits); // Outputs: ['banana', 'orange']
console.log(firstFruit); // Outputs: 'apple'
```

```
[ 'banana', 'orange' ]
apple
```

4. unshift()

```
let fruits = ['apple', 'banana'];
let newLength = fruits.unshift('pineapple', 'mango');
console.log(fruits); // Outputs: ['pineapple', 'mango', 'apple', 'banana']
console.log(newLength); // Outputs: 4
```

```
[ 'pineapple', 'mango', 'apple', 'banana' ]
4
```

Arrays and Array Methods Example

5. map()

```
let numbers = [1, 2, 3, 4];
let squares = numbers.map(num => num * num);
console.log(squares); // Outputs: [1, 4, 9, 16]
```

```
[ 1, 4, 9, 16 ]
```

6. filter()

```
let numbers = [1, 2, 3, 4, 5];
let evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers); // Outputs: [2, 4]
```

```
[ 2, 4 ]
```

Arrays and Array Methods Example

7. reduce()

```
let numbers = [1, 2, 3, 4];
let sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
console.log(sum); // Outputs: 10
```

```
PS C:\Users\Anushka\Downloads\react\node.js> node reduce.js
10
```

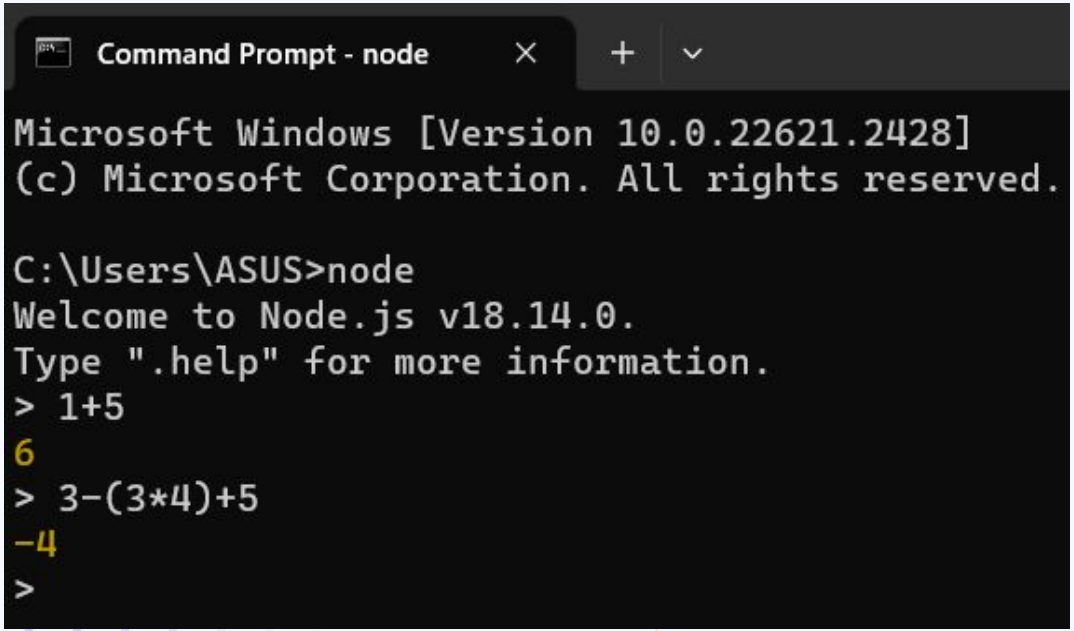
REPL Terminal

Node comes bundled with a REPL environment like Windows console or Unix/Linux shell.

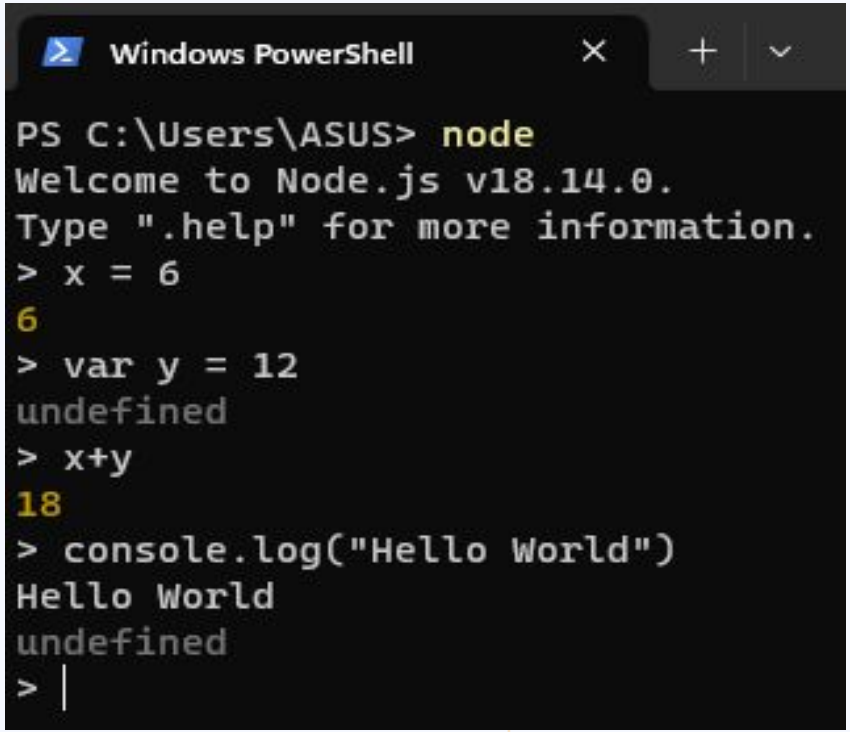
- **Read** - Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- **Eval** - Takes and evaluates the data structure
- **Print** - Prints the result
- **Loop** - Loops the above command until the user presses ctrl+c twice.

Starting REPL

Simply run **node** on shell/console without any arguments.



Use Variables



Multiline Expression

```
> var x = 0
undefined
> do {
... x++
... console.log("x:" +x);
... }
... while(x<5);
x:1
x:2
x:3
x:4
x:5
undefined
>
```

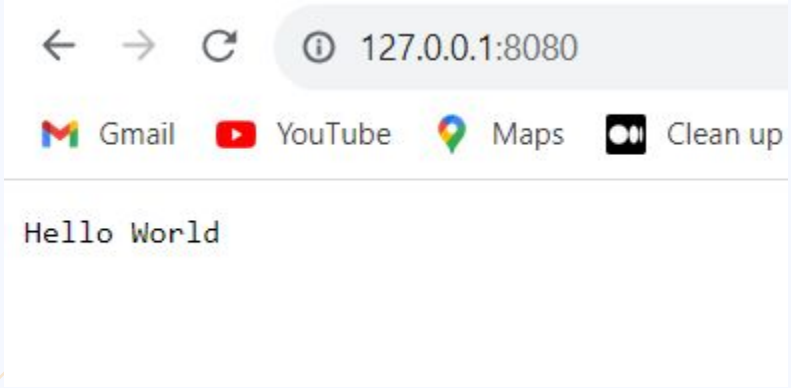
Web-based Node.js Application

- A web-based Node.js application consists of the following important components:
  - Import required modules
    - Lode Node.js modules using the require directive
  - Create server
    - Create a server to listen the client's requests
  - Read request and return response
    - Read the client request made using browser or console and return the response.

firstwebapp.js

```
var http = require('http');

http.createServer(function(req, res){
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(8080);
```



Web-based Node.js Application ctd.

- createServer() method turns your computer into an HTTP server.
- listen() method makes the server listen to ports on the computer.

Request a text file from the server

```
var http = require('http')
var url = require('url')
var fs = require('fs')

http.createServer(function(req, res) {
    var pathName = url.parse(req.url).pathname
    fs.readFile(pathName.substring(1), function(err, data) {
        console.log(pathName.substring(1))
        if(err) {
            res.writeHead(404, {'Content-Type':'text/html'})
            console.log(err)
        } else {
            res.writeHead(200, {'Content-Type':'text/html'})
            res.write(data)
            //console.log(data)
        }
        res.end()
    })
}).listen(8080);
console.log('server running');
```

Different Response Methods

1. **res.writeHead**
   This method sends a response header to the request. The statusCode is a 3-digit HTTP status code, such as 200 for a successful request or 404 for Not Found with the type of the response
1. **res.write**
   This method is used to send a chunk of the response body. This can be called multiple times to provide successive parts of the body.
   Eg : res.write('</body>');
   res.write('</html>');
   res.end(); // No more data to write, finish the response.
1. **res.end**
   This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete

JavaScript Callbacks

Suppose you want to do a calculation, and then display the result.

Two ways we can simply without callbacks,

1. **Save the result from a function, and then call another function**

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2) {
  let sum = num1 + num2;
  return sum;
}

let result = myCalculator(5, 5);
myDisplayer(result);
```

2. **Let the calculator function call the display function (myDisplayer):**

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some
}

function myCalculator(num1, num2) {
  let sum = num1 + num2;
  myDisplayer(sum);
}

myCalculator(5, 5);
```

JavaScript Callbacks Contd

A callback is a function passed as an argument to another function.

```
function myDisplayer(display) {
  document.getElementById("demo").innerHTML = display;
}

function myDisplayer2(result) {
  console.log("The result is: " + result);
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
myCalculator(3, 3, myDisplayer2);
```

Asynchronous Operations

```
function printResult(result) {
  console.log("The result is: " + result);
}
function calculateSumAsync(num1, num2, callback) {
  console.log("Calculation started, please wait...");
  setTimeout(() => {
    let sum = num1 + num2;
    callback(sum); // The callback function is called once the operation is complete.
  }, 2000); // 2-second delay
}
console.log("Script start");
calculateSumAsync(5, 7, printResult);
console.log("Script end - The calculation is happening asynchronously.");
```

```
Script start
Calculation started, please wait...
Script end - The calculation is happening asynchronously.
The result is: 12
```

Callbacks Concept

- Callbacks Concept is called when a task is completed, thus helping preventing any kind of blocking or nonblocking
- Callbacks Concept nonblocking allows other code to run in the meantime.
- Using the Callback concept, Node.js can process a large number of requests without waiting for any function to return the result which makes Node.js highly scalable.

Callbacks Concept

Blocking Code Example

- main.js > ...

```
1  var fs = require("fs");
2  var data = fs.readFileSync('input.txt');
3
4  console.log(data.toString());
5  console.log("Program ended");
```

```
PS D:\CSE Course> node main.js
This is an example text file created for MERN Course
Program ended
PS D:\CSE Course>
```

Callbacks Concept ctd.

Non - Blocking Code Example

- ```
  var fs = require("fs");

  fs.readFile('input.txt', function(err,data){
      if(err) return console.error(err);
      console.log(data.toString());
  });

  console.log("Program ended");
  ```

```
PS D:\CSE Course> node nonblocking.js
Program ended
This is an example text file created for MERN Course
PS D:\CSE Course>
```
```

Event Loop

- Node.js is a single-threaded event-driven platform that is capable of running non-blocking, asynchronous programming.
- These functionalities of Node.js make it memory efficient.
- The event loop allows Node.js to perform non-blocking I/O operations despite the fact that JavaScript is single-threaded.
- It is done by assigning operations to the operating system whenever and wherever possible.

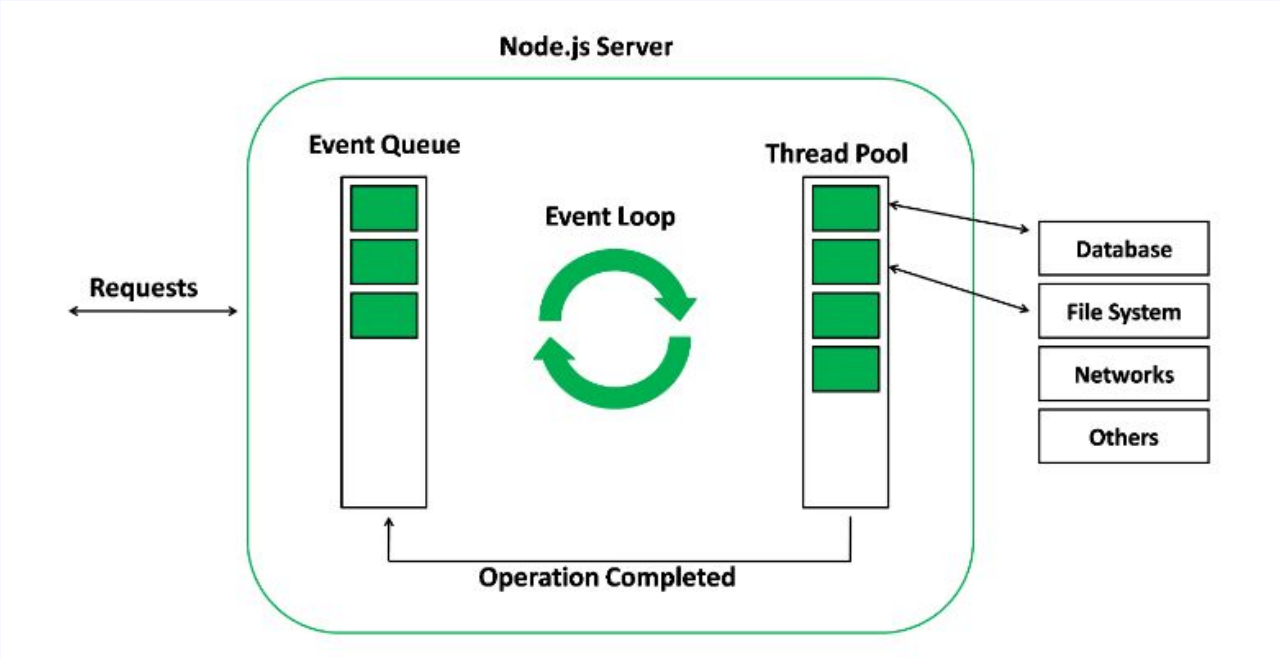
Event Loop ctd.

- Most operating systems are multi-threaded and hence can handle multiple operations executing in the background.
- When one of these operations is completed, the kernel tells Node.js, and the respective callback assigned to that operation is added to the event queue which will eventually be executed.

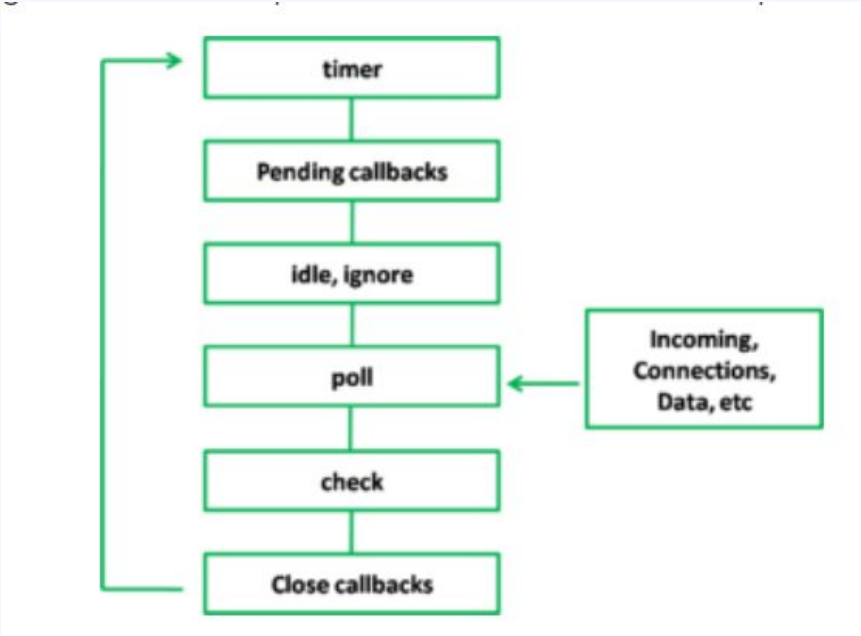
Features of Event Loop

- An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more tasks.
- The event loop executes tasks from the event queue only when the call stack is empty i.e. there is no ongoing task.
- The event loop allows us to use callbacks and promises.
- The event loop executes the tasks starting from the oldest first.

Working of the event loop



Phases of Event Loop



Timers


JS timers.js > ...
1 console.log('Start');
2
3 setTimeout(() => {
4 console.log('Timeout callback');
5 }, 2000);
6
7 console.log('End');

Poll

```
const fs = require('fs');

console.log('Start');

fs.readFile('input.txt', (err, data) => {
    if (err) throw err;
    console.log('File Read Callback');
  });

setImmediate(function() {
    console.log('Immediate Callback');
});
console.log('End');
```

Check

Even though the check phase should be running right after poll phase, Since the poll is empty at first check phase will be running, so we need to handle it with promises.

```
const fs = require('fs');

console.log('Start');

fs.readFile('input.txt', function(err, data){
    if(err) return console.error(err);
    console.log(data.toString());
});

setImmediate(function() {
    console.log('Immediate Callback');
});

console.log('End');
```

Promises in Node.js

- Callback functions are used for Asynchronous events.
- Whenever any asynchronous event has to be taken place it is generally preferred to use callbacks .
    - **if data is not nested or inter-dependent
- A **promise** is basically an advancement of callbacks in Node. In other words, a promise is a JavaScript object which is used to handle all the asynchronous data operations.

Promise Object Properties

- JavaScript Promise object can be:
  - Pending
  - Fulfilled
  - Rejected
- The Promise object supports two properties: state and result.
- While a Promise object is "pending" (working), the result is undefined.
- When a Promise object is "fulfilled", the result is a value.
- When a Promise object is "rejected", the result is an error object.

Example with Promises

```
const fs = require("fs");

// Util function to promisify fs.readFile
function readFileAsync(path) {
  return new Promise((resolve, reject) => {
    fs.readFile(path, (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}
```

```
console.log("Start");

// Read file within a promise
readFileAsync("input.txt")
  .then((data) => {
    console.log(data.toString());
    // After the file read is completed, setImmediate is called.
    setImmediate(() => {
      console.log("Immediate Callback");
    });
  })
  .catch((err) => {
    console.error(err);
  })
  .finally(() => {
    console.log("End");
  });
```

Async/Await

- Async and Await make promises easier to write. By using async and await, you can write asynchronous code that looks and behaves a bit more like synchronous code, which can make it easier to understand and maintain.
- An **async function** returns a promise, and the resolved value of the promise will be whatever you return from the function.
- **Await** can be used inside async functions to pause the execution of the function until the Promise is resolved. When you await a promise, the function is paused in a non-blocking way, meaning the JavaScript engine can execute other tasks while waiting for the promise to resolve.

Example with Async/Await

```
const fs = require('fs').promises; // Node.js v10+

// Async function to read a file
async function readFileAsync(path) {
  try {
    console.log('Start reading file...');

    const data = await fs.readFile(path, 'utf8');
    console.log(data);

    console.log('File read successfully.');
  } catch (err) {
    console.error('An error occurred:', err);
  }
}

// Call the async function
readFileAsync('input.txt');
```

Callbacks, Promises and Async/Await

Callbacks – Callbacks are functions passed as arguments to other functions. They are commonly used in asynchronous operations to specify what should happen once a task is completed.

Promises – Promises provide a cleaner and more structured way to handle asynchronous operations. A Promise represents a value that might be available now, or in the future, or never.

Async/Await – The async/await syntax is built on top of Promises and provides a more concise and synchronous-looking way to work with asynchronous code. Asynchronous code is essential for handling tasks that involve waiting, such as I/O operations, network requests, and interactions with databases, without blocking the entire program's execution. It enables more efficient utilization of resources and better responsiveness in applications.

Event Driven Programming

- Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies.
- As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.
- The functions that listen to events act as Observers.
- Whenever an event gets fired, its listener function starts executing.

JS eventlisten.js > ...

```
// Importing events
const EventEmitter = require('events');

// Initializing event emitter instances
var eventEmitter = new EventEmitter();

var listener1= (msg) => {
    console.log("Message from listener1: " + msg);
};

var listener2 = (msg) => {
    console.log("Message from listener2: " + msg);
};

// Registering listener1 and listener2

eventEmitter.on('myEvent', listener1);
eventEmitter.on('myEvent', listener1);
eventEmitter.on('myEvent', listener2);

// Removing listener1
eventEmitter.removeListener('myEvent', listener1);

// Triggering myEvent
eventEmitter.emit('myEvent', "Event occurred");

// Removing all the listeners to myEvent
eventEmitter.removeAllListeners('myEvent');

// Triggering myEvent
eventEmitter.emit('myEvent', "Event occurred");
```

Buffers

- The Buffer class in Node.js is used to perform operations on raw binary data.
- Generally, Buffer refers to the particular memory location in memory.
- Buffer and array have some similarities
- But,
  - Array can be any type, and it can be resizable.
  - Buffers only deal with binary data, and it can not be resizable.
- Each integer in a buffer represents a byte.

buffers.js > ...

```
// Different methods to create buffers
const buffer1 = Buffer.alloc(100);  // Create a buffer of 100 bytes
const buffer2 = Buffer.from('UCSC');  // Create a buffer from a string
const buffer3 = Buffer.from([1, 2, 3, 4]);  // Create a buffer from an array
// const buffer4 = new Buffer("Has an Issue with this");

// Writing data to a buffer
buffer1.write("Happy Learning");

// Reading data from a buffer
const a = buffer1.toString('utf-8');
console.log(a);

// Checking if an object is a buffer
console.log(Buffer.isBuffer(buffer1));

// Getting the length of a buffer
console.log(buffer1.length);

// Copying a buffer
const bufferSrc = Buffer.from("ABC");
console.log(bufferSrc.toString('utf-8'));
const bufferDest = Buffer.alloc(3);
bufferSrc.copy(bufferDest);
console.log(bufferDest.toString('utf-8'));

// Slicing data
const bufferOld = Buffer.from('Hello world');
const bufferNew = bufferOld.slice(0, 4);
console.log(bufferNew.toString());

// Concatenating two buffers
const bufferOne = Buffer.from('Happy Learning');
const bufferTwo = Buffer.from(' With UCSC');
const bufferThree = Buffer.concat([bufferOne, bufferTwo]);
console.log(bufferThree.toString());
```

Streams

Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

4 types of Streams in Node.js

- Readable - Stream which is used for read operation.
- Writable - Stream which is used for write operation.
- Duplex - Stream which can be used for both read and write operation.
- Transform - A type of duplex stream where the output is computed based on input.

Streams ctd.

- Each type of Stream is an EventEmitter instance and throws several events at different instance of times.
- Commonly used events
  - **data** - This event is fired when there is data is available to read.
  - **end** - This event is fired when there is no more data to read.
  - **error** - This event is fired when there is any error receiving or writing data.
  - **finish** - This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

```
var fs = require("fs");
var data = '';

// Create a readable stream
var readerStream = fs.createReadStream('input.txt');

// Set the encoding to be utf8.
readerStream.setEncoding('UTF8');

// Handle stream events --> data, end, and error
readerStream.on('data', function(chunk) {
   data += chunk;
});
```

```
readerStream.on('end',function() {
   console.log(data);
});

readerStream.on('error', function(err) {
   console.log(err.stack);
});

console.log("Program Ended");
```

Writing to a Stream

writingstream.js > ...

```
1   var fs = require("fs");
2   var data = 'Simply Easy Learning';
3
4   // Create a writable stream
5   var writerStream = fs.createWriteStream('output.txt');
6
7   // Write the data to stream with encoding to be utf8
8   writerStream.write(data,'UTF8');
9
10  // Mark the end of file
11  writerStream.end();
12
13  // Handle stream events --> finish, and error
14  writerStream.on('finish', function() {
15      console.log("Write completed.");
16  });
17
18  writerStream.on('error', function(err) {
19      console.log(err.stack);
20  });
21
22  console.log("Program Ended");
```

Global Objects

- Node.js Global Objects are the objects that are available in all modules.
- Global Objects are built-in objects that are part of the JavaScript and can be used directly in the application without importing any particular module.

List of Global Objects

- __dirname
- __filename
- Console
- Process
- Buffer
- setImmediate(callback[, arg][, ...])
- setInterval(callback, delay[, arg][, ...])
- setTimeout(callback, delay[, arg][, ...])
- clearImmediate(immediateObject)
- clearInterval(intervalObject)
- clearTimeout(timeoutObject)

Thanks!